Chapter 8: Object-Oriented Databases

- New Database Applications
- The Object-Oriented Data Model
- Object-Oriented Languages
- Persistent Programming Languages
- Persistent C++ Systems
New Database Applications

• Data models designed for data-processing-style applications are not adequate for new technologies such as computer-aided design, computer-aided software engineering, multimedia and image databases, and document/hypertext databases.

• These new applications requirement the database system to handle features such as:
  – complex data types
  – data encapsulation and abstract data structures
  – novel methods for indexing and querying
Object-Oriented Data Model

- Loosely speaking, an *object* corresponds to an entity in the E-R model.
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into a single unit.
- The object-oriented data model is a logical model (like the E-R model).
- Adaptation of the object-oriented programming paradigm (e.g., Smalltalk, C++) to database systems.
Object Structure

- An object has associated with it:
  - A set of *variables* that contain the data for the object. The value of each variable is itself an object.
  - A set of *messages* to which the object responds; each message may have zero, one, or more *parameters*.
  - A set of *methods*, each of which is a body of code to implement a message; a method returns a value as the *response* to the message
- The physical representation of data is visible only to the implementor of the object
- Messages and responses provide the only external interface to an object.
Messages and Methods

- The term message does not necessarily imply physical message passing. Messages can be implemented as procedure invocations.

- Methods are programs written in a general-purpose language with the following features
  - only variables in the object itself may be referenced directly
  - data in other objects are referenced only by sending messages

- Strictly speaking, every attribute of an entity must be represented by a variable and two methods, e.g., the attribute address is represented by a variable address and two messages get-address and set-address.
  - For convenience, many object-oriented data models permit direct access to variables of other objects.
Object Classes

- Similar objects are grouped into a class; each such object is called an *instance* of its class.
- All objects in a class have the same
  - variable types
  - message interface
  - methods

They may differ in the values assigned to variables.

- Example: Group objects for people into a *person* class.
- Classes are analogous to entity sets in the E-R model.
Class Definition Example

class employee {
    /* Variables */
    string name;
    string address;
    date start-date;
    int salary;
    /* Messages */
    int annual-salary();
    string get-name();
    string get-address();
    int set-address(string new-address);
    int employment-length();
};

- For strict encapsulation, methods to read and set other variables are also needed
- employment-length is an example of a derived attribute
Inheritance

- E.g., class of bank customers similar to class of bank employees: both share some variables and messages, e.g., name and address. But there are variables and messages specific to each class e.g., salary for employees and and credit-rating for customers.

- Every employee is a person; thus employee is a specialization of person.

- Similarly, customer is a specialization of person.

- Create classes person, employee and customer
  - variables/messages applicable to all persons associated with class person.
  - variables/messages specific to employees associated with class employee; similarly for customer.
Inheritance (Cont.)

- Place classes into a specialization/IS-A hierarchy
  - variables/messages belonging to class *person* are *inherited* by class *employee* as well as *customer*
- Result is a class hierarchy

Note analogy with ISA hierarchy in the E-R model
Class Hierarchy Definition

class person {
    string name;
    string address;
};
class customer isa person {
    int credit-rating;
};
class employee isa person {
    date start-date;
    int salary;
};
class officer isa employee {
    int office-number;
    int expense-account-number;
};
...
Class Hierarchy Example (Cont.)

- Full variable list for objects in the class officer:
  - office-number, expense-account-number: defined locally
  - start-date, salary: inherited from employee
  - name, address: inherited from person

- Methods inherited similar to variables.

- Substitutability — any method of a class, say person, can be invoked equally well with any object belonging to any subclass, such as subclass officer of person.

- class extent: set of all objects in the class. Two options:
  1. Class extent of employee includes all officer, teller and secretary objects
  2. Class extent of employee includes only employee objects that are not in a subclass such as officer, teller or secretary
Example of Multiple Inheritance

Class DAG for banking example.

```
person

employee

full-time  part-time  teller  secretary

officer  full-time-teller  part-time-teller  full-time-secretary  part-time-secretary
```
Multiple Inheritance

- The class/subclass relationship is represented by a directed acyclic graph (DAG) — a class may have more than one superclass.
- A class inherits variables and methods from all its superclasses.
- There is potential for ambiguity. E.g., variable with the same name inherited from two superclasses. Different solutions such as flag and error, rename variables, or choose one.
- Can use multiple inheritance to model “roles” of an object.
  - A person can play the roles of student, a teacher or footballPlayer, or any combination of the three (e.g., student teaching assistants who also play football).
  - Create subclasses such as student-teacher and student-teacher-footballPlayer that inherit from multiple classes.
Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.

- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
  - Value – data value; used in relational systems.
  - Name – supplied by user; used for variables in procedures.
  - Built-in – identity built into data model or programming language.
    * no user-supplied identifier is required.
    * form of identity used in object-oriented systems.
Object Identifiers

- **Object identifiers** used to uniquely identify objects
  - can be stored as a field of an object, to refer to another object.
  - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
  - can be system generated (created by database) or external (such as social-security number)
Each component in a design may contain other components.

Can be modeled as containment of objects. Objects containing other objects are called *complex* or *composite* objects.

Multiple levels of containment create a *containment hierarchy*: links interpreted as *is-part-of*, not *is-a*.

Allows data to be viewed at different granularities by different users.
Object-Oriented Languages

- Object-oriented concepts can be used as a design tool, and be encoded into, for example, a relational database (analogous to modeling data with E-R diagram and then converting to a set of relations).

- The concepts of object orientation can be incorporated into a programming language that is used to manipulate the database.
  - Object-relational systems – add complex types and object-orientation to relational language.
  - Persistent programming languages – extend object-oriented programming language to deal with databases by adding concepts such as persistence and collections.
Persistent Programming Languages

- Persistent programming languages:
  - allow objects to be created and stored in a database without any explicit format changes (format changes are carried out transparently).
  - allow objects to be manipulated in-memory – do not need to explicitly load from or store to the database.
  - allow data to be manipulated directly from the programming language without having to go through a data manipulation language like SQL.

- Due to power of most programming languages, it is easy to make programming errors that damage the database.

- Complexity of languages makes automatic high-level optimization more difficult.

- Do not support declarative querying very well.
Persistence Of Objects

- Approaches to make transient objects persistent include establishing persistence by:
  - Class – declare all objects of a class to be persistent; simple but inflexible.
  - Creation – extend the syntax for creating transient objects to create persistent objects.
  - Marking – an object that is to persist beyond program execution is marked as persistent before program termination.
  - Reference – declare (root) persistent objects; objects are persistent if they are referred to (directly or indirectly) from a root object.
Object Identity and Pointers

- A persistent object is assigned a persistent object identifier.
- Degrees of permanence of identity:
  - Intraprocedure – identity persists only during the execution of a single procedure
  - Intraprogram – identity persists only during execution of a single program or query.
  - Interprogram – identity persists from one program execution to another.
  - Persistent – identity persists throughout program executions and structural reorganizations of data; required for object-oriented systems.
Object Identity and Pointers (Cont.)

- In O-O languages such as C++, an object identifier is actually an in-memory pointer.

- Persistent pointer – persists beyond program execution; can be thought of as a pointer into the database.
How to find objects in the database:

- Name objects (as you would name files) – cannot scale to large number of objects.
  - typically given only to class extents and other collections of objects, but not to objects.

- Expose object identifiers or persistent pointers to the objects – can be stored externally.
  - All objects have object identifiers.
How to find objects in the database (Cont):

- Store collections of objects and allow programs to iterate over the collections to find required objects.
  - Model collections of objects as *collection types*
  - *Class extent* – the collection of all objects belonging to the class; usually maintained for all classes that can have persistent objects.
Persistent C++ Systems

- C++ language allows support for persistence to be added without changing the language
  - Declare a class called Persistent_Object with attributes and methods to support persistence
  - Overloading – ability to redefine standard function names and operators (i.e., +, −, the pointer dereference operator ->) when applied to new types
- Providing persistence without extending the C++ language is
  - relatively easy to implement
  - but more difficult to use
ODMG C++ Object Definition Language

- Standardize language extensions to C++ to support persistence.
- ODMG standard attempts to extend C++ as little as possible, providing most functionality via template classes and class libraries.
- Template class `Ref<class>` used to specify references (persistent pointers).
- Template class `Set<class>` used to define sets of objects. Provides methods such as `insert_element` and `delete_element`.
- The C++ object definition language (ODL) extends the C++ type definition syntax in minor ways. Example: Use notation `inverse` to specify referential integrity constraints.
class Person : public Persistent_Object {
public:
    String name;
    String address;
};

class Customer : public Person {
public:
    Date member_from;
    int customer_id;
    Ref<Branch> home_branch;
    Set<Ref<Account>> accounts inverse Account::owners;
};
class Account : public Persistent_Object {
private:
    int balance;
public:
    int number;
    Set<Ref<Customer>> owners inverse Customer::accounts;
    int find_balance();
    int update_balance(int delta);
};
ODMG C++ Object Manipulation Language

- Uses persistent versions of C++ operators such as `new(db)`.
  
  ```
  Ref<Account> account = new(bank_db) Account;
  ```
  `new` allocates the object in the specified database, rather than in memory

- Dereference operator `->` when applied on a `Ref<Customer>` object loads the referenced object in memory (if not already present) and returns in-memory pointer to the object.

- **Constructor** for a class – a special method to initialize objects when they are created; called automatically when `new` is executed

- **Destructor** for a class – a special method that is called when objects in the class are deleted
```cpp
int create_account_owner(String name, String address) {
    Database * bank_db;
    bank_db = Database::open("Bank-DB");
    Transaction Trans;
    Trans.begin();
    Ref<Account> account = new(bank_db) Account;
    Ref<Customer> cust = new(bank_db) Customer;
    cust->name = name;
    cust->address = address;
    cust->accounts.insert_element(account);
    account->owners.insert_element(cust);
    // ... Code to initialize customer_id, account_number etc.
    Trans.commit();
}
```
int print_customers() {
    Database * bank_db;
    bank_db = Database::open("Bank-DB");
    Transaction Trans;
    Trans.begin();
    Iterator<Ref<Customer>> iter =
        Customer::all_customers.create_iterator();
    Ref<Customer> p;
    while (iter.next(p)) {
        print_cust(p);
    }
    Trans.commit();
}

- Iterator construct helps step through objects in a collection.